Computers & Geosciences 36 (2010) 171–178

Contents lists available at ScienceDirect

Computers & Geosciences

journal homepage: www.elsevier.com/locate/cageo

Parallel drainage network computation on CUDA

L. Ortega *, A. Rueda

Departamento de Informática, Edif. A3-140, Campus de Las Lagunillas, Universidad de Jaén, 23071 Jaén, Spain

ARTICLE INFO

Article history:
Received 11 July 2008
Received in revised form
16 July 2009
Accepted 17 July 2009

Keywords:
GPU
GPGPU
CUDA
Drainage network
D8 algorithm

ABSTRACT

Drainage networks determination from digital elevation models (DEM) has been a widely studied problem in the last three decades. During this time, satellite technology has been improving and optimizing digitalized images, and computers have been increasing their capabilities to manage such a huge quantity of information. The rapid growth of CPU power and memory size has concentrated the discussion of DEM algorithms on the accuracy of their results more than their running times.

However, obtaining improved running times remains crucial when DEM dimensions and their resolutions increase. Parallel computation provides an opportunity to reduce run times. Recently developed graphics processing units (GPUs) are computationally fast not only in Computer Graphics but in General Purpose Computation, the so-called GPGPU. In this paper we explore the parallel characteristics of these GPUs for drainage network determination, using the C-oriented language of CUDA developed by NVIDIA. The results are simple algorithms that run on low-cost technology with a high performance response, obtaining CPU improvements of up to 8×.

1. Introduction

Digital elevation models (DEMs) provide useful information about the morphology of a terrain surface, including hydrology. Considering a DEM as a grid model whose cells represent the elevation of a square surface, the main principle in hydrology can be established: water flows down the steepest slope on the surface. Extracting a digital representation of the flow network is considered an essential step in the study of watershed delineation, erosion sites, mineral or pollution distribution, the cost estimation and design of constructing new roads, the simulation of floodplains in paddy fields (Ishida et al., 2003), etc.

Most of the methods we find in the literature are based on grids. In comparison with triangulated networks (TINs) or contour maps, grid models have become more important as computer capabilities have increased at the same time that memory cost has decreased. In the early nineties grid dimensions were a few hundreds of thousands of cells, while nowadays most personal computers are capable of handling hundreds of millions of cells. In any case, massive grid terrains must be partitioned into squared cell areas as in Curkendall et al. (2003) and Arge et al. (2001b), or into watersheds for computing the watershed graph as in Arge et al. (2001a). The partial solutions are processed in parallel and linked together to create a complete solution.

In many cases, traditional methods for obtaining drainage networks are still extensively used because of their simplicity and reduced limitations, as pointed out in Martz and Garbrecht (1992). There are mainly two hydrological approaches derived from raster DEMs based on flow accumulation. The first was introduced by O'Callaghan and Mark (1984) using a neighborhood of eight cells as possible flow directions, the denominated *D*8 model. In order to obtain a channel network, a threshold value should be defined. Those cells exceeding this threshold are part of a drainage channel. The method also considers some processes previous to the drainage accumulation: (1) an optional smoothing phase to reduce artificial pits generated by the data acquisition system; (2) the drainage direction assignment to establish which of the eight neighboring cells receives the accumulated water; (3) the labeling of point drainage features as pits, ridges or forks; (4) a special process for removing pits assuming that water finds overflow points from the basin of a pit. After that, an iterative process simulates how each cell $c(i,j)$ drains into a neighbor cell while some other cell (or cells) may drain into $c(i,j)$.

The results of applying this method are sometimes non-realistic in divergent areas where eight directions seem to be insufficient. Different procedures but similar results are found in Martz and Jong (1988) and Jenson and Dominique (1988). The main difference between these two approaches lies in the way that sinks are interpreted. While in Martz and Jong (1988) all sinks are assumed to be real topographic features which should be treated hydrologically as ponds or reservoirs, Jenson and Dominique (1988) approach assumes that the sinks are primarily data errors or artifacts.

In addition, a previous morphological study should be performed in order not to impose an arbitrary and spatially constant drainage density, or avoid several executions of the flow

* Corresponding author. Tel.: +34 953 212890; fax: +34 953 212472.
E-mail addresses: lidia@ujaen.es (L. Ortega), ajrueda@ujaen.es (A. Rueda).

0098-3004/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cageo.2009.07.005

network algorithm to determine the threshold value. These methods can determine the appropriate drainage density at which to extract networks from digital elevation data (see Tarboton et al., 1991), or even can be adaptive to spatial variability in drainage density by identifying and connecting upwards curved grid cells as in Tarboton and Ames (2001).

The second approach, introduced by Freeman (1991), can be considered more sophisticated: the drainage of a particular cell flows downslope to several adjacent cells of lower elevation; the divergent flow concept. The resulting drainage network improves the *D*8 model in some aspects, but needs additional computational time and obtains wide flooding areas rather than narrow channels.

GIS applications are an important area for parallel computation because of the huge amount of information handled, the complexity of spatial algorithms and the requirement of a realtime response for many operations (see Healey et al., 1997). Conversion between vector and raster formats, overlay operations, visibility computation on terrains and of course, drainage network determination, are suitable operations for efficient parallel implementations. Mower (1993) proposed a parallel solution for drainage network computation using the method of O'Callaghan and Mark (1984) and implemented it for old parallel hardware from Thinking Machines. He focused on the subproblem of drainage basin labeling, however, the drainage accumulation phase, by far the most time consuming, was not given in detail.

We also adopted O'Callaghan and Mark (1984) approach for our parallel solution using GPUs. This method is simpler, is able to generate connected networks and is the most commonly used because it relies on a runoff analog to define the flow paths. Another approach could achieve to better results under specific morphological terrain surfaces, but our main goal is to compare running times using CPU versus GPU technology under the same conditions. We assume that these results can be extended to many other methods for channel network delineation as well as to many other DEM applications. The matrix data structure of grid DEM and the identical and repetitive procedure performed in each cell make this problem fit perfectly with a GPGPU resolution.

NVIDIA CUDA technology is being introduced in many different research areas. In this work we use CUDA for grid DEM algorithms in order to solve a standard flooding problem. In Section 2 we define some concepts about GPGPU using CUDA. In Section 3 we analyze two different strategies using CUDA to compute a drainage network. Next, in Section 4 we provide computation times of GPU versus CPU to reach to some interesting conclusions in Section 5 about the benefits of GPGPU programming using CUDA in DEM models.

2. CUDA technology for GPGPU applications

Algorithm 1. *void CUDA_Procedure* (*CPUdataStructures data*)

```
1: Define and reserve memory for GPU data structures
2: Copy data from CPU to GPU global memory
3: Define numThreads and numBlocks variables
4: Execute CUDA procedure for each thread
5: Copy data results from GPU to CPU
6: Free GPU data structures
```

The general-purpose computing on graphics processing units (GPGPU) is a young area of research that has attracted attention of many research groups in the last years. Although graphics hardware has been used for general-purpose computation since the 1970s, the flexibility, power processing and low cost of modern graphics processing units (GPUs) have generalized its use for solving many problems in Signal Processing, Computer Vision, Computational Geometry or Scientific Computing (Owens et al., 2007).

A GPU can be programmed by defining vertex, pixel or geometry shaders (Akenine-Möller et al., 2008) in a specific shader language like GLSL (Rost, 2006), HLSL or Cg (Fernando and Kilgard, 2003). A shader is a program that implements a special processing for a specific graphics primitive during the rendering process. For instance, a vertex shader can be used to compute the position of the vertex over time and implement complex animations; in a similar way, diverse lighting effects can be coded in a pixel shader to simulate specific materials like skin or hair (Fernando, 2004; Pharr and Fernando, 2005). In the last years, shader programming has been extensively used by graphics programmers and has dramatically improved the visual quality and realism of video games.

Using shader programming for GPGPU is a tricky process that implies stating a general problem as a graphic computation, and therefore can be difficult in many cases. The rigid memory model is the biggest problem: GPU architecture only allows memory reads from textures or a limited set of input parameters. In Compute Graphics, textures are typically color images that are mapped onto 3D surfaces to improve realism (Akenine-Möller et al., 2008), but in shader programming they can code other input information useful for the computation. For similar reasons memory writes can only be performed on a fixed position in the framebuffer. This often makes that a simple computation that can be coded with a few lines in a CPU program could require a complete redesign and a careful implementation in GPU. On the positive side, the implementation effort is usually rewarded with a superb performance, up to 100× faster than equivalent CPU implementations in some cases.

The CUDA architecture of NVIDIA (2008) represents a major advance for the development of GPGPU applications. For the first time, a GPU can be used without any knowledge of computer graphics, as a general-purpose highly parallel coprocessor that helps the CPU in the more complex and time-expensive computations. With CUDA a GPU can be programmed in C, in a very similar style to a CPU implementation, and the memory model is now simpler and more flexible.

A CUDA-enabled GPU is composed of several MIMD multiprocessors (Multiple Instruction stream, Multiple Data stream) that enclose a set of SIMD processors (Single Instruction stream, Multiple Data stream). Each multiprocessor has a small shared memory that can be accessed from each of its processors, and there is a large global memory space common to all the multiprocessors (Fig. 1). Shared memory is very fast and is usually used for caching data from global memory. Both shared and global memory can be accessed by any thread for reading and writing operations without restrictions.

A CUDA execution runs several blocks of threads. Each thread performs a single computation and is executed by a SIMD processor. A block is a set of threads that are executed on the same multiprocessor and its size should be chosen to maximize the use of the multiprocessor. A thread can store data on its local registers, share data with other threads from the same block through the shared memory or access the device global memory. The number of blocks usually depends on the amount of data to process. Each thread is assigned a local index inside the block with three components, starting at (0, 0, 0), although in most cases only one component (x) is used. The blocks are indexed using a similar scheme.

A typical CUDA computation follows the steps shown in Algorithm 1. The host function starts by allocating one or more buffers in the GPU global memory and transfers the data to